**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе №3**

**по дисциплине «Объектно-ориентированное программирование»**

Тема: «Контейнеры вектор и список»

|  |  |  |
| --- | --- | --- |
| Студент гр. 7381 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Дорох С.В. |
| Преподаватель | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Жангиров Т.М. |

Санкт-Петербург

2019

**Цель работы.**

Изучить стандартные контейнеры vector и list языка С++.

**Задание.**

Необходимо реализовать конструкторы, деструктор, оператор присваивания, функции assign, resize, erase, insert и push\_back для контейнера вектор (в данном уроке предполагается реализация упрощенной версии, без резервирования памяти под будущие элементы).

Поведение реализованных функций должно быть таким же, как у класса std::vector.

Необходимо реализовать список со следующими функциями: вставка элементов в голову и в хвост, получение элемента из головы и из хвоста, удаление из головы и из хвоста, очистка, проверка размера, деструктор, конструктор копирования, конструктор перемещения, оператор присваивания.

Также необходимо реализовать итератор для списка. Для краткости реализации можно ограничиться однонаправленным изменяемым (неконстантным) итератором. Необходимо реализовать операторы: =, ==, !=, ++ (постфиксный и префиксный), \*, ->.

С использованием итераторов необходимо реализовать вставку элементов (вставляет value перед элементом, на который указывает pos; возвращает итератор, указывающий на вставленный value), удаление элементов (удаляет элемент в позиции pos; возвращает итератор, следующий за последним удаленным элементом).

Поведение реализованных функций должно быть таким же, как у класса std::list. Семантику реализованных функций нужно оставить без изменений.

При выполнении этого задания можно определять любые вспомогательные функции. Вводить или выводить что-либо не нужно. Реализовывать функцию main не нужно. Не используйте функции из cstdlib (malloc, calloc, realloc и free).

**Ход работы.**

Был реализован класс vector; поведение реализованных функций аналогично поведению функций класса std::vector.

Класс vector содержит два поля: указатели на начало и конец массива данных в памяти. Были реализованы деструктор и следующие конструкторы: конструктор от размера массива, от двух итераторов, от списка инициализации, копирования и перемещения. Также были реализованы методы изменения размера, удаления одного элемента или интервала элементов, вставки одного элемента или нескольких элементов, заданных при помощи двух итераторов, на заданное итератором место и вставки одного элемента в конец вектора.

Реализация класса представлена в приложении А.

Класс list имеет аналогичные поля, как и у класса vector, но данные содержатся не в массиве, а в двусвязном списке. Для класса list были реализованы деструктор и следующие конструкторы: стандартный, копирования и перемещения. Также был реализованы оператор присваивания и методы для вставки, получения и удаления элементов из головы и из хвоста, очистки списка и проверки размера. Поведение реализованных функций аналогично поведению функций класса std::list.

Итератор для списка содержит одно поле – указатель на элемент контейнера list. Для итератора был перегружен ряд операторов: =, ==, !=, ++ (постфиксный и префиксный), \* и ->. Класс list объявлен в данном классе, как дружественный, так как используется в функциях для вставки и удаления элементов из списка.

Реализация класса представлена в приложении А.

**Вывод.**

В ходе выполнения лабораторной работы была изучена реализация контейнеров vector и list.

**Приложение А. Файл vector.h.**

#include <assert.h>

#include <algorithm> // std::copy, std::rotate

#include <cstddef> // size\_t

#include <initializer\_list>

#include <stdexcept>

namespace stepik

{

template <typename Type>

class vector

{

public:

typedef Type\* iterator;

typedef const Type\* const\_iterator;

typedef Type value\_type;

typedef value\_type& reference;

typedef const value\_type& const\_reference;

typedef std::ptrdiff\_t difference\_type;

explicit vector(size\_t count = 0) : m\_first(count ? new value\_type[count] : nullptr), m\_last(m\_first+count)

{}

template <typename InputIterator>

vector(InputIterator first, InputIterator last) : m\_first(new value\_type[last-first]), m\_last(m\_first+(last-first))

{

std::copy(first, last, m\_first);

}

vector(std::initializer\_list<Type> init) : m\_first(new value\_type[init.size()]), m\_last(m\_first+init.size())

{

std::copy(init.begin(), init.end(), m\_first);

}

vector(const vector& other) : m\_first(new value\_type[other.size()]), m\_last(m\_first+other.size())

{

std::copy(other.begin(), other.end(), m\_first);

}

vector(vector&& other) : m\_first(other.m\_first), m\_last(other.m\_last)

{

other.m\_first = nullptr;

other.m\_last = nullptr;

}

~vector()

{

delete[] m\_first;

}

void resize(size\_t count)

{

if(count == size())

return;

vector Temp(count);

std::copy(m\_first, count > size() ? m\_last : m\_first+count, Temp.m\_first);

std::swap(m\_first, Temp.m\_first);

std::swap(m\_last, Temp.m\_last);

}

vector& operator=(const vector& other)

{

delete[] m\_first;

m\_first = new value\_type[other.size()];

m\_last = m\_first + other.size();

std::copy(other.begin(), other.end(), m\_first);

}

vector& operator=(vector&& other)

{

delete[]m\_first;

std::swap(m\_first, other.m\_first);

std::swap(m\_last, other.m\_last);

other.m\_first = nullptr;

other.m\_last = nullptr;

}

// assign method

template <typename InputIterator>

void assign(InputIterator first, InputIterator last)

{

delete[] m\_first;

m\_first = new value\_type[last-first];

m\_last = m\_first+(last-first);

std::copy(first, last, m\_first);

}

//insert methods

iterator insert(const\_iterator pos, const Type& value)

{

//vector Temp(size() + 1);

size\_t offset = pos - m\_first;

//std::copy(m\_first, m\_first+offset, Temp.m\_first);

//Temp.m\_first[offset] = value;

//std::copy(m\_first+offset, m\_last, Temp.m\_first+offset+1);

resize(size()+1);

std::rotate(m\_first+offset, m\_last - 1, m\_last);

\*(m\_first+offset) = value;

//std::swap(m\_first, Temp.m\_first);

//std::swap(m\_last, Temp.m\_last);

return m\_first+offset;

}

template <typename InputIterator>

iterator insert(const\_iterator pos, InputIterator first, InputIterator last)

{

size\_t count = last - first;

size\_t offset = pos - m\_first;

resize(size() + count);

std::rotate(m\_first+offset, m\_last - count, m\_last);

std::copy(first, last, m\_first+offset);

return m\_first+offset;

}

iterator erase(const\_iterator pos)

{

vector Temp(size()-1);

size\_t offset = pos - m\_first;

std::copy(m\_first, m\_first+offset, Temp.m\_first);

std::copy(m\_first+(offset+1), m\_last, Temp.m\_first+offset);

std::swap(m\_first, Temp.m\_first);

std::swap(m\_last, Temp.m\_last);

return m\_first+offset;

}

iterator erase(const\_iterator first, const\_iterator last)

{

size\_t offset = last - first;

size\_t begin = first - m\_first;

vector Temp(size() - offset);

std::copy(m\_first, m\_first+begin, Temp.m\_first);

std::copy(m\_first+(begin+offset), m\_last, Temp.m\_first+begin);

std::swap(\*this,Temp);

return m\_first+begin;

}

//push\_back methods

void push\_back(const value\_type& value)

{

vector Temp(size() + 1);

std::copy(m\_first, m\_last, Temp.m\_first);

Temp.m\_first[size()] = value;

std::swap(m\_first, Temp.m\_first);

std::swap(m\_last, Temp.m\_last);

}

//at methods

reference at(size\_t pos)

{

return checkIndexAndGet(pos);

}

const\_reference at(size\_t pos) const

{

return checkIndexAndGet(pos);

}

//[] operators

reference operator[](size\_t pos)

{

return m\_first[pos];

}

const\_reference operator[](size\_t pos) const

{

return m\_first[pos];

}

//\*begin methods

iterator begin()

{

return m\_first;

}

const\_iterator begin() const

{

return m\_first;

}

//\*end methods

iterator end()

{

return m\_last;

}

const\_iterator end() const

{

return m\_last;

}

//size method

size\_t size() const

{

return m\_last - m\_first;

}

//empty method

bool empty() const

{

return m\_first == m\_last;

}

private:

reference checkIndexAndGet(size\_t pos) const

{

if (pos >= size())

{

throw std::out\_of\_range("out of range");

}

return m\_first[pos];

}

//your private functions

private:

iterator m\_first;

iterator m\_last;

};

}

**Приложение Б. Файл list.h.**

#include <assert.h>

#include <algorithm>

#include <stdexcept>

#include <cstddef>

#include <utility>

namespace stepik

{

template <class Type>

struct node

{

Type value;

node\* next;

node\* prev;

node(const Type& value, node<Type>\* next, node<Type>\* prev)

: value(value), next(next), prev(prev)

{

}

};

template <class Type>

class list; //forward declaration

template <class Type>

class list\_iterator

{

public:

typedef ptrdiff\_t difference\_type;

typedef Type value\_type;

typedef Type\* pointer;

typedef Type& reference;

typedef size\_t size\_type;

typedef std::forward\_iterator\_tag iterator\_category;

list\_iterator()

: m\_node(NULL)

{

}

list\_iterator(const list\_iterator& other)

: m\_node(other.m\_node)

{

}

list\_iterator& operator = (const list\_iterator& other)

{

m\_node = other.m\_node;

return \*this;

}

bool operator == (const list\_iterator& other) const

{

return m\_node == other.m\_node;

}

bool operator != (const list\_iterator& other) const

{

return m\_node != other.m\_node;

}

reference operator \* ()

{

return m\_node->value;

}

pointer operator -> ()

{

return &(m\_node->value);

}

list\_iterator& operator ++ ()

{

m\_node = m\_node->next;

return \*this;

}

list\_iterator operator ++ (int)

{

list\_iterator\* tmp = \*this;

m\_node = m\_node->next;

return tmp;

}

private:

friend class list<Type>;

list\_iterator(node<Type>\* p)

: m\_node(p)

{

}

node<Type>\* m\_node;

};

template <class Type>

class list

{

public:

typedef Type value\_type;

typedef value\_type& reference;

typedef const value\_type& const\_reference;

typedef list\_iterator<Type> iterator;

list()

: m\_head(nullptr), m\_tail(nullptr)

{

}

~list()

{

clear();

}

list::iterator begin()

{

return iterator(m\_head);

}

list::iterator end()

{

return iterator();

}

void push\_back(const value\_type& value)

{

if(!m\_head) {

m\_head = new node<Type>(value, nullptr, nullptr);

m\_tail = m\_head;

}

else {

node<Type>\* new\_node = new node<Type>(value, nullptr, m\_tail);

m\_tail->next = new\_node;

m\_tail = new\_node;

}

}

void push\_front(const value\_type& value)

{

if(!m\_head) {

m\_head = new node<Type>(value, nullptr, nullptr);

m\_tail = m\_head;

}

else {

node<Type>\* new\_node = new node<Type>(value, m\_head, nullptr);

m\_head->prev = new\_node;

m\_head = new\_node;

}

}

reference front()

{

return m\_head->value;

}

const\_reference front() const

{

return m\_head->value;

}

reference back()

{

return m\_tail->value;

}

const\_reference back() const

{

return m\_tail->value;

}

void pop\_front()

{

if(!empty()) {

if(size() == 1) {

delete m\_head;

m\_head = nullptr;

m\_tail = nullptr;

}

else {

m\_head = m\_head->next;

delete m\_head->prev;

m\_head->prev = nullptr;

}

}

}

void pop\_back()

{

if(!empty()) {

if(size() == 1) {

delete m\_head;

m\_head = nullptr;

m\_tail = nullptr;

}

else {

m\_tail = m\_tail->prev;

delete m\_tail->next;

m\_tail->next = nullptr;

}

}

}

void clear()

{

while (m\_head)

pop\_back();

}

bool empty() const

{

return !m\_head;

}

size\_t size() const

{

node<Type>\* tmp = m\_head;

int count = 0;

while(tmp) {

count++;

tmp = tmp->next;

}

return count;

}

iterator insert(iterator pos, const Type& value)

{

if(pos == begin()) {

push\_front(value);

return begin();

}

if(pos.m\_node == nullptr) {

push\_back(value);

return iterator(m\_tail);

}

pos.m\_node->prev->next = new node<Type>(value, pos.m\_node, pos.m\_node->prev);

pos.m\_node->prev = pos.m\_node->prev->next;

return pos.m\_node->prev;

}

iterator erase(iterator pos)

{

if(pos == begin()) {

pop\_front();

return begin();

}

if(pos.m\_node == nullptr) {

return pos;

}

if(pos.m\_node->next == nullptr) {

pop\_back();

return end();

}

node<Type>\* tmp = pos.m\_node->next;

pos.m\_node->prev->next = pos.m\_node->next;

pos.m\_node->next->prev = pos.m\_node->prev;

delete pos.m\_node;

return tmp;

}

private:

//your private functions

node<Type>\* m\_head;

node<Type>\* m\_tail;

};

}